Reinforcement Learning Based SDN Controller Load Balancing

A Project Report Submitted

for the Course

MA 499 Project II

by

AB Satyaprakash

(Roll No. 180123062)

to the

DEPARTMENT OF MATHEMATICS

INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI

GUWAHATI - 781039, INDIA

April 2022

CERTIFICATE

This is to certify that the work contained in this report entitled **"SDN Controller Load Balancing based on Reinforcement Learning"** submitted by **AB Satyaprakash** (**Roll No: 180123062**) to Department of Mathematics, Indian Institute of Technology Guwahati towards the partial requirement of Bachelor in Technology in Mathematics and Computing has been carried out by him under my supervision.

It is also certified that, in addition to the literature survey, the student has established new results, performed computational implementations, and conducted empirical analysis as part of the project.

Turnitin Similarity: 11%

Guwahati - 781039 (Dr. Ashok Singh Sairam)

April 2022 Project Supervisor

ABSTRACT

Software-Defined Network (SDN) is a networking paradigm featuring the separation of the control plane from the data plane of the network devices. SDN offers distinct advantages over traditional networks due to its enhanced centralized management and network programmability. The control plane is physically distributed but logically centralized to meet the issue of scalability. Since network traffic is both spatially and temporally dynamic, a multi-controller organization in SDN needs a load balancing procedure to deal with local overloads effectively.
The primary motivation behind this work is to provide a framework that learns from the network traffic and seeks to balance it quickly, decrease the unnecessary *migration cost* and improve the *in-packet request-response rate*. To this end, we use the load ratio deviation between the controllers and generate optimal migration triplets, consisting of the migrate-out and migration-in domains and a set of switches for migration. To obtain the global optimal controller load balancing with the lowest cost, we utilize an *online Q-learning* with the constraints of maximum efficiency and no migration conflicts. Our work has shown good results in a variety of scenarios with different load distribution on switches. Using simulation results we demonstrate the convergence behavior of the proposed scheme to the optimal policy.

Contents

List of Figures **vi**

List of Tables **vii**

1 Introduction **1**

1.1 Organization of The Report 3

2 Background and Literature Survey **4**

2.1 Overview of Software-defined Network & Reinforcement Learning 4

2.1.1 Software-defined Network (SDN): 4

2.1.2 Reinforcement Learning (RL): 6

2.2 Surveys 7

3 Switch Migration Design **9**

3.1 Load Balancing 10

3.1.1 Controller's Load (L_{C_i}) 10

3.1.2 Controller's Load Ratio (L_{C_i}) 11

3.1.3 Discrete Coefficient (D) 11

3.2 Switch Migration Efficiency 12

4 System Model **13**
4.1 Assumptions 13
4.2 State and Action 15
4.2.1 State Space ($\mathscr{S}$) 15
4.2.2 Action Space ($\mathscr{A}$) 15
4.2.3 Transitions and Transition probability 16
4.3 Reward and Costs 17
4.3.1 Reward ($\mathscr{R}$) 17
4.3.2 Cost ($\mathscr{C}$) 17

5 Problem Formulation **19**

6 Methodology **22**
6.1 Random Approach 22
6.2 Greedy Approach 23
6.3 Q-learning with ε-greedy 25
6.3.1 Proposed Q-learning algorithm 25
6.3.2 Q-learning implementation 26

7 Results and Implementation **29**
7.1 Experiment Setup 29
7.2 Evaluation Metric: 31
7.3 Experimental Results 31

8 Conclusion & Future Work **43**

Bibliography **44**

List of Figures

4.1 An example of state transitions with incoming and outgoing flows 16

5.1 A state-action example for the MDP formulation. 20

5.2 Optimal migration model based on reinforcement learning 21

7.1 Example of a sample data table 30

7.2 Discrete coefficient of the system with time - Load Heavy 34

7.3 Cumulative discounted reward of the system with time - Load Heavy 35

7.4 Number of switches exchanged with time - Load Heavy 36

7.5 Discrete coefficient of the system with time - Load Light 37

7.6 Cumulative discounted reward of the system with time - Load Light 38

7.7 Number of switches exchanged with time - Load Light 39

7.8 Discrete coefficient of the system with time - Load Skewed 40

7.9 Cumulative discounted reward of the system with time - Load Skewed 41

7.10 Number of switches exchanged with time - Load Skewed 42

List of Tables

3.1 List of symbols for parameters in load balancing 10

4.1 List of symbols for describing problem statement in Q-learning . . 14

Chapter 1

Introduction

Software-Defined Networking brings the unique advantage of decoupling the control and the data plane of networks. From the viewpoint of an application, the network appears as a centralized control plane. This architecture makes design and operation of the network simple. It also makes the network programmable and allows for the network resources optimization. One or more physical controllers can be used to achieve the SDN's centralised control plane. For a multi-controller setup, the controllers synchronize their local network messages to form a global view. Although theoretically, one physical controller implements SDN in all aspects, this creates some serious scalability issues for more complex networks. When dealing with a huge number of requests, a controller's limited resources can become a bottleneck. Furthermore, using a single controller exposes the network to a single point of failure.

Researchers have developed a distributed SDN architecture to alleviate the issues associated with a single controller. The goal is evenly distributing the network's load among all of the controllers. Moreover, if a controller fails, its load should be ideally distributed among the remaining controllers, thus making the architecture scalable and robust. In this context, researchers have introduced the *controller placement problem* (CPP) since the network performance of a distributed SDN architecture depends on the layout of the controllers.

In [8, 10], authors have studied of work on controller placement problem in SDN by researchers. In these papers, CPP in SDN is mostly divided into based on either performance matrices or network traffic i.e., static or adaptive (dynamic) controller placement problem. In scenario of static controller placement, main concern of researchers is optimal placement of the controllers to control the set of switches based on one-time consideration of the performance metrics. In adaptive controller placement, the initial placement of controllers are based on a particular performance metric. Thereafter, the assignment of the switches to controllers are changed dynamically in time or space in accordance with variation in network traffic. Thus balancing the controller load basically boils down to the problem of switch migration.

We discovered that switching a switch is based on historical or present traffic conditions during our examination of the state of the art on switch migration. However, researchers have discovered that for large time scales (hours or weeks), Internet traffic has a seasonal trend, whereas for small time scales, it is highly chaotic (seconds or minutes). As a result, methods for learning traffic behaviour are being developed, and then a controller is being chosen, and the switches are being reassigned to that controller. This method will show to be far more advantageous. Learning the traffic of genuine SDN networks would be the biggest problem here. The issue of Internet traffic modelling is still a work in progress. For a variety of MDP-modeled situations, the reinforcement learning (RL) [11] algorithm is a well-studied machine learning approach. A learning agent senses the environment and takes an action at each state to complete a well-defined goal and maximise the collected action rewards in reinforcement learning. The RL algorithm has the advantage of being able to make time-series decisions in a discrete space while evolving action rules, which is important for load balancing problems in SDN, according to its learning capabilities. Thus, our main objective of this work is to use reinforcement learning [11] to learn the traffic behavior to balance the load of the network in a dynamic environment. We make use of an online ε-greedy based Q-learning which makes the reward cost optimisation along with the exploration and exploitation strategy. Based on the result, we can claim that our method improves load balancing in the dynamic environment of SDN.

1.1 Organization of The Report

This chapter gives a quick overview of our work, which will be addressed later in the study. Chapter 2 provides some context and related work on balancing the controller load in software-defined networking and summarizes the problems that were not addressed in that approach. Chapter 3 discusses about the switch migration design and formation of the overloaded and under loaded domains. The 4th chapter delves into the models and their brief discussion while 5th chapter focuses on the problem formulation. The 6th chapter provides the methodology of our work. The result and implementation details are presented in Chapter 7. Chapter 8 summarises the findings from Chapter 7 and discusses potential work based on the findings.

Chapter 2

Background and Literature Survey

This chapter has two parts. In the first part we discuss the basics of SDN and RL, while the second part will consist of surveys related to load balancing and reinforcement learning.

2.1 Overview of Software-defined Network & Reinforcement Learning

2.1.1 Software-defined Network (SDN):

SDN (software-defined networking) is an emerging network paradigm that advocates separating a network's control plane from its data plane to maximize resource efficiency. SDN technology is being used in more areas than traditional networks, such as the IoT, Next-Generation Future Networks (NGFN), and 5G. For example, Google's B4 data center, which spans the globe, has implemented Software-Defined WAN.

Architecture of SDN: The architecture of SDN comprises three layers - the *infrastructure layer*, which comprises forwarding devices such as switches or routers; the *control layer*, which consists of a controller as decision-maker; and finally, the *application layer*. A controller, which manages network intelligence, is at the heart of the SDN. A southbound API is used to communicate between switches (forwarding devices) and controllers, with OpenFlow being a well-known example. The control plane and the application layer communicate via the northbound API. A single controller may not be sufficient in extensive networks, necessitating the use of numerous controllers. EastWest API is used as an interface among the multiple controllers. A comprehensive survey of SDN networks is given in [7].

Controller placement problem in SDN: In SDN, a single controller has performance, scalability, and single-point failure restrictions. Multiple controller deployment was designed to handle the limits of a single controller, boost performance, scalability, reliability, and fault tolerance. Multiple controller placements present two significant challenges: the count and placement of the deployed controllers, including another issue: synchronizing controllers while keeping in mind SDN features, as each controller having a global network view.

The controller placement problem has already sparked much research. The issue can be divided into two categories: *static controller placement* and *adaptive controller placement*. In static controller placement, the switch-controller mapping is statically defined based on the initial traffic conditions. This method ignores the fact that traffic loads change over time. The mapping between a switch and a controller can change with changes in traffic load over time in an adaptive controller placement problem.

The adaptive controller placement problem addresses temporal variation in traffic patterns by dynamically reassigning switches to the controller. On the other hand, reassigning the switches ensures that network performance measures such as flow setup time, migration cost, load balance, and fault tolerance are not harmed.

2.1.2 Reinforcement Learning (RL):

Reinforcement Learning is a decision-making algorithm. It is a machine learning (ML)-based training strategy that rewards positive activities while punishing undesired ones. This optimal behavior is acquired through interactions with the environment and watching how it reacts. It is comparable to how kids examine their surroundings and pick up on the habits that help them achieve their goals.

In RL, the learning agent is trained to assign positive values to beneficial acts and negative values to counterproductive actions. Accordingly, there are both positive and negative reinforcement types. To obtain an optimal solution, the agent is motivated to seek long-term and most significant overall gain. Long-term goals also keep the agent from getting stuck on minor objectives while learning through contact with the environment. In reinforcement learning all decisions are made sequentially. This means that the output of the state depends on the current input and the next input depends on the previous input. Since the learning here is dependent, we give labels to sequences of dependent decisions.

2.2 Surveys

In this section we will look at the research already done in the field of Load Balancing in SDN and for Reinforcement Learning.

Both temporal and geographical traffic fluctuations exist in real networks. Temporal traffic fluctuations are determined by the hour of the day, and spatial traffic variations are caused by the flows generated by applications connected with distinct switches.

The cumulative traffic at a controller may surpass its capacity as a result of these fluctuations. To address this load imbalance, we may need to transfer switches from overloaded to underloaded controllers. The migrations, unlike static CPP, are done on the fly. As a result, we must handle the issue of interrupted flows.

Dixit et al.[3] [4] proposed to address the issue by using the *equal* controller mode (specified in *OpenFlow v*1.2) while transitioning a controller from master to slave. They also propose expanding or contracting the controller pool depending on whether the controller load jumps the upper or lower threshold.

Wang et al. [13] proposed a framework *Switch Migration-based Decision-Making* (SMDM) where they compute *load diversity*, a ratio of controller loads. If the load diversity between two controllers exceeds a certain level, switch migration occurs. The switches selected for migration are those with less load and higher efficiency.

Hu et al. [6] proposed *Efficiency-Aware Switch Migration* (EASM), where they measure the degree of load balancing using the normalized load variance of the controller load. In the event of the load difference matrix exceeding a threshold, the controller load is presumed to be imbalanced. The threshold is a function of the difference between the maximum and minimum controller load.

Filali et al. [5] use the ARIMA time series model to predict switch's load. The forecasting allows finding the time step at which a controller will become overloaded and accordingly schedule a switch migration in advance. The authors use a predetermined threshold to identify overloaded controllers.

The work by Zhou et al. [14] are among those few who consider the problem of *load oscillation* due to inappropriate switch migration. The issue of load oscillation is where underloaded controllers used to offload the traffic load of overloaded controllers, themselves become quickly overloaded.

Ul Haque et al. [12] address to balance the variation in the controller load by employing a controller module, which is a set of controllers. Their method estimate the number of flows that the switches will produce on a regular interval and accordingly activates the appropriate number of controllers.

Chen et al. [2] use a game-theoretic method to solve the problem of controller load balancing. The underloaded controllers are modelled as players who compete for switches from overloaded controllers. The payoffs are determined when an underloaded controller is selected as the master controller of a victim switch.

Sangho et al. [9] propose the switch-aware reinforcement learning load balancing (SAR-LB) for achieving balanced Load Distribution with Reinforcement Learning-Based Switch Migration in Distributed SDN Controllers.

After this survey, we find that system become unbalanced due to change in network traffic. If we learn behavior of network traffic then we can take right decision. In this study, we use reinforcement learning for creating a framework that learns network traffic patterns and can make the best selection for selecting the underloaded controller that provides loadbalancing.

Chapter 3

Switch Migration Design

In this chapter, we discuss the switch migration design. Researchers have discussed that by appropriate switch migration, the overload trend among the controllers can be reduced, thus enabling the existing controllers to manage their resources effectively to handle the requests from the switches. This section introduces several terminologies: controller's load and load ratios, migration efficiency, and discrete coefficients of controllers. The selection algorithms are designed to generate migration tuples, consisting of the out-migration domain, the in-migration domain, and a set of migration switches, which will also be used in the reinforcement learning process discussed in the subsequent chapters. Some important symbols, that are used in this chapter (and also later in the report), is described in Table 3.1.

3.1 Load Balancing

The first task at hand balancing the controllers' load. In this section we discuss 3 parameters, namely,load, load ratio, and discrete coefficient.

Table 3.1 List of symbols for parameters in load balancing

Symbol	Description
L_{Ci}	The load of the controller C_i
C_{Ci}	The load capacity of the controller C_i
R_{Ci}	The load ratio of the controller C_i
L_{Sk}	The packet-in message sending rate of the switch S_k
R	Mean load ratio of all controllers
D	Load ratio deviation coefficient between two controllers
E	Migrate switch efficiency between controllers
H_{Sk}	Number of hops between the switch and the controller

3.1.1 Controller's Load ($\mathbf{L}_{C_i}$)

- The primary source of a controller's load comes from the of packet in messages it receives from it's switches.

- Thus we can say that the instantaneous load on a controller is the sum of its packet-in message rates, i.e,

$$L_{C_i} = \sum_{k=1}^{n} L_{S_k} \tag{3.1}$$

3.1.2 Controller's Load Ratio (L$_{C_i}$)

- Based on criteria such as CPU performance, number of processors, memory size, and so on, controllers might have varying load capacities.
- The load ratio for a controller is defined as the ratio of the controller load to its load capacity. Obviously the mean load ratio of the controllers is the mean of the load ratios for the controllers.

$$\begin{cases} R_{C_i} = L_{C_i}/C_{C_i}, \\ \overline{R} = \sum_{i=1}^{n} R_{C_i}/n \end{cases} \tag{3.2}$$

3.1.3 Discrete Coefficient (D)

- The discrete coefficient is a measure of the load balancing between the controllers. It is defined as the deviation between the controllers' load ratio and the mean load ratio between the controllers.
- For 2 controllers, the discrete coefficient can be defined as:

$$D_{(C_i,C_j)} = \left(\sqrt{\sum_{k=i,j} \left(R_{C_k} - \overline{R_{C_i,C_j}}\right)^2 / 2} \right) / \overline{R_{C_i,C_j}}, \tag{3.3}$$

- When we consider multiple controllers the discrete coefficient becomes -

$$D_{(C_i,C_j)} = \left(\sqrt{\sum_{i=1}^{n} \left(R_{C_k} - \overline{R}\right)^2 / 2} \right) / \overline{R}, \tag{3.4}$$

- The discrete coefficient of all controllers is one of the indicators for evaluating the performance of the selection algorithm. A small value of discrete coefficient indicates a better load balancing of the controller plane.

3.2 Switch Migration Efficiency

We will have a migration overload, if a switch is migrating between controllers. The migration efficiency is represented as follows because the hop count H_{S_k} between the switch and the controller is different, as is the load L_{S_k} of the switch to the controller.

$$E_{(C_i,S_k)} = L_{S_k}/H_{S_k}, \tag{3.5}$$

Chapter 4

System Model

In this chapter, we first describe the load balancing problem, then formulate our problem using Markov Decision Process (MDP). In our case we will consider these 4 parameters - **state**, **action**, **reward** and **cost**.

4.1 Assumptions

We consider a system where a switch is assigned to or controlled by a controller at a time t. Thus,

$$x_{ij}(t) = \begin{cases} 1, & if \text{ a link exists b/w switch } s_i \text{ and} \\ & \text{controller } c_j \text{ at time } t, \\ 0, & otherwise \end{cases} \tag{4.1}$$

Since we are in a multi-controller setup, a controller can be mapped to zero or more switches. Every switch has a limited capability of processing flows. We assume that the maximum number of flows supported by each switch is N_F.

The number of new flow arrivals at switch s_i in time t is denoted by $s_i(t)$. A controller's load is defined as the total number of flows that reached to it at time t. The instantaneous load of a controller c_j at time t can be computed as below.

$$L_{c_j}(t) = \sum_{i=1}^{n} s_i(t)x_{ij}(t) \tag{4.2}$$

In this report, we consider the ideal load of controller as mean load of system i.e.,

$$L^I(t) = \frac{\sum_{j=1}^{K} L_{c_j}(t)}{K} \tag{4.3}$$

We want to minimize the difference between ideal load which calculated by equation 4.3 and instantaneous load which computed by equation 4.2. For this purpose we introduce the discrete coefficient given by equation 3.4 and use it as a decision factor to determine the quality of our implementation.

Table 4.1 List of symbols for describing problem statement in Q-learning

Symbol	Description
S	Set of switches, $\mid S \mid = n$
C	Set of controllers, $\mid C \mid = k$
$s(t)$	Flow-request rate of switch s at time t
$L_c(t)$	Instantaneous load of controller c at time t
$L^{\mathrm{I}}(t)$	Ideal load of all the controller at time t

4.2 State and Action

We assume that for each of the switches S_i, the flows arrivals can be modelled with a Poisson process with parameter λ_i. The service times are considered to be exponentially distributed with means $\frac{1}{\mu_i}$. The arrival of a new flow, or the departure of a serviced request is taken as a decision epoch. Since the system at hand has a **Markovian** nature, we observe the system state at each of the decision epoch. We don't need to consider the system state at any other time points except for the decision epoch because the system does not change elsewhere.

4.2.1 State Space ($\mathscr{S}$)

- In our problem, the state space $\mathscr{S}$ is a set of tuples consisting of the number of flows in each controller.
- Each state $ST \in \mathscr{S}$ corresponds to a tuple described by $\{$number of flows in C_1, number of flows in C_2, ..., number of flows in $C_k\}$

4.2.2 Action Space ($\mathscr{A}$)

- At every decision epoch, an action chosen at the current state moves the system to the next state. Since at every decision epoch only one controller C_i, either gets or loses a flow, we define the action $a \in \mathscr{A}$ for that controller.
- Action space for C_i is $\{$do nothing, add a switch to C_i, remove a switch from $C_i\}$

4.2.3 Transitions and Transition probability

As the system moves from state s to a state s', we can define a transition probability for the same. From each state-action pair (s,a), the system moves to a different state s' with finite probability $p_{ss'}(a)$. We will later consider Q-Learning which is a model-free algorithm, but for the sake of understanding let's consider an example.

Let's say we have 2 controllers and 3 switches. From our assumption we have N_F as the maximum number of flows a switch can support. Let's consider that $N_F = 2$. Under the action *do nothing* we have -

$$p_{ss'}(a) = \frac{\lambda_1}{\sum_{i=1}^{3} \lambda_i + 6\mu} \tag{4.4}$$

where $s = (i,j)$ and $s' = (i+1, j)$.

In the Fig. 4.1, we can look at some state transitions happening in a system with 5 controllers at 3 random time points. Note that along side actions that we are taking, the state is also being modified at each time point with the incoming flows or departure of the serviced flows.

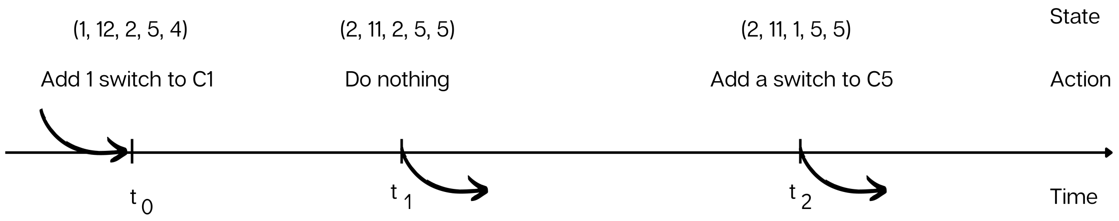


Fig. 4.1 An example of state transitions with incoming and outgoing flows

4.3 Reward and Costs

When a system is at the state s, and some action a is chosen, the reward has to be some fairness metric which has to be decided upon in a way that it reflects the long-term outcome of the action. Similarly for every action we take, we incur some overhead which we have to define as a cost. For every switch migration between the controllers, we have to make sure that the cost is minimized. We define these terms as follows:

4.3.1 Reward ($\mathscr{R}$)

The reward is taken to be negative of the discrete coefficient, i.e.,

$$r(s,a) = -D_{ij} \tag{4.5}$$

This is reasonable since discrete coefficient is a representation of the degree of disorder in the system of controller loads. If we take an action a that increases the disturbance, then the reward must be more negative. On the other hand an action that stabilises the system must receive a smaller negative reward.

4.3.2 Cost ($\mathscr{C}$)

If no action is taken then no cost should be incurred. On the other hand for every switch exchange, flows are redirected to the new controller which is the reason for an overhead. We define the cost function as follows -

$$c(s,a) = \begin{cases} 0, & if \text{ the action is do nothing} \\ \text{number of flows} & \\ \text{exchanged between controllers,} & otherwise \end{cases} \tag{4.6}$$

Chapter 5

Problem Formulation

Based on our thorough discussion in the last chapter, we can conclude that we aim to reduce the discrete coefficient of the system to as low as possible, while keeping in mind that we don't create huge migration overheads or conflicts. We can model the system in terms of rewards and thus we can restate the first part of the objective as maximising the total discounted reward for the system. Since the arrival and departure of flows occurs at any time this makes it a continuous time problem. Therefore, this problem can be formulated as a constrained MDP.

Objective: Maximise the total discounted reward over the infinite horizon subject to total discounted cost over the infinite horizon. Let S_{max} is the constraint on average number of switch exchanges between controllers, the formulated constrained MDP can be given as -

$$\begin{cases} \max_{\pi \in U} \lim_{T\to\infty} \left(\sum_{t=0}^{T-1} \alpha^t E_\pi[r(s_t, a_t)]\right) \\ \text{subject to } \lim_{T\to\infty} \left(\sum_{t=0}^{T-1} \alpha^t E_\pi[c(s_t, a_t)]\right) \leq S_{max} \end{cases} \tag{5.1}$$

Let's take a scenario where we have 13 switches which are distributed under 3 controllers namely c_1, c_2 and c_3 as depicted in Fig. 5.1. Consider at a particular time step t, c_2 is overloaded. Now to balance the controller c_2 a random switch s_4 is selected for migration, if the action of **remove a switch** is chosen by the ε-greedy strategy. Let c_1 be the most under loaded controller for the current iteration. s_4 is now moved to c_1. Migration of switch is shown in Fig 5.1.

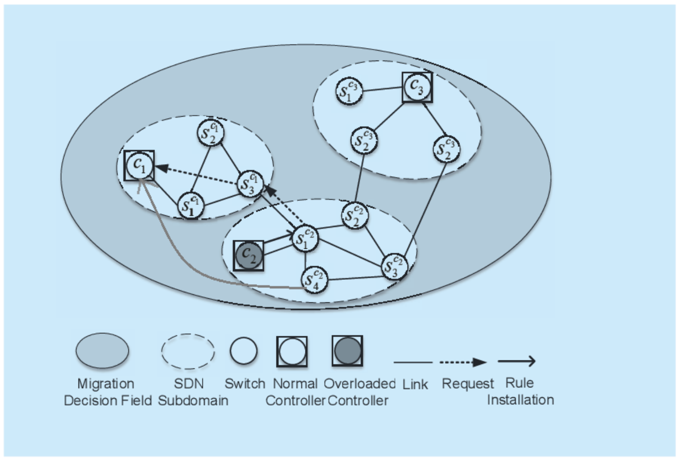


Fig. 5.1 A state-action example for the MDP formulation.

The decision-making process in reinforcement learning, which is defined as taking action depending on the present state to acquire the return value, is the main driving force for optimising the migration model. Figure 5.2 depicts the optimal switch migration model.

By selecting the largest item in each row of the ε-greedy strategy, the value of $Q(ST,a)$ is iteratively changed. The following is an example of an iteratively updated valuation matrix method. In this report, however, we will use a slightly modified version of the below Q-learning algorithm to account for the constraints on the cost incurred.

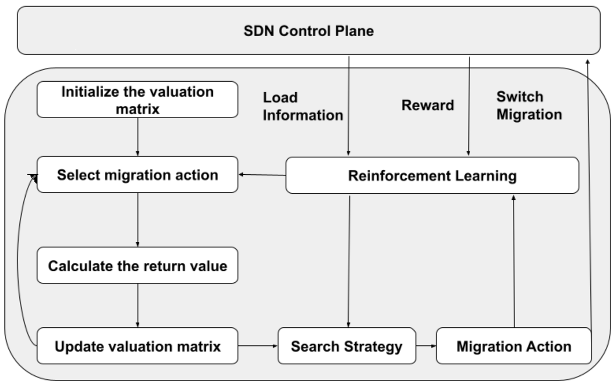


Fig. 5.2 Optimal migration model based on reinforcement learning

$$Q(ST,a)_{t+1} = Q(ST,a)_t + \alpha(R(ST,a) + \gamma * \max_a Q(ST',a')_t - Q(ST,a)_t) \quad (5.2)$$

The expected value of the valuation is calculated as the sum of the present return value γ and the next biggest maximum value $\max_a Q(ST,a)_t$ in the memory. The difference between the expected expectation and the exact estimate is used to obtain the value of $Q(ST,a)_t$ in the $(t+1)^{th}$ round of incremental, iterative learning. The learning rate α is set between 0 and 1, with 0 indicating no updates to the Q-values. As a result, nothing is learnt, and a high number, such as 0.9, indicates that learning can happen quickly. Furthermore, the larger the magnitude of the discount factor γ, the more long-term benefits are considered, and the smaller the size of the immediate consideration.

Chapter 6

Methodology

In this section, we propose three ways for a load balancing mechanism. The first one is by selecting the switch and target controller randomly, and the second is a complete greedy approach that works by generating tuples for migration. The third one works by using an on-line Q-learning algorithm and makes use of a Q-table.

6.1 Random Approach

For each time step, we calculate the ideal load and then classify the controllers in I_{domain} and O_{domain}. One of the overloaded controller $c_i \in O_{domain}$ is selected and undergoes load balancing by migrating a switch s_j randomly from $c_i.s$ to target controller which is selected randomly. This process continues until it gets balanced.

6.2 Greedy Approach

For a pure greedy implementation, we proceed in two steps. First, we divide the controller sets into two domains, i.e., out and in-migration domains. Since the controller load changes fast in a real-time changing environment, collisions in the control plane are common. The mechanism we use for migration domain selection ensures that collisions are avoided by assessing each controller appropriately. This decision is made considering the average load rate and discrete coefficients presented in the previous chapter.

Algorithm 1: Selection Algorithm of Migration Domain

```
input  : $R = \{R_{C1}, R_{C2}, ..., R_{Cn}\}$: Controller's load ratio
         $\overline{R}$: Mean load ratio of all controllers
         $D$: Discrete Coefficient between the controllers
output: Out-migration domain O, in-migration domain I, migration queue Q
initialize: Set O, I and Q to empty sets
begin
1   foreach $R_{Ci}$, $R_{Cj}$ $\in$ $R=\{R_{C1}, R_{C2}, ..., R_{Cn}\}$ do
2       $D_{C_i,C_j}$: Calculate the discrete coefficients
3       if $D_{C_i,C_j} > D$ and $R_{Ci} > \overline{R}$ and $R_{Ci} > R_{Cj}$ then
4           Add Ci to the out-migration domain O
5           Add Cj to the in-migration domain I
6       if $D_{C_i,C_j} < D$ and $R_{Ci} < \overline{R}$ and $R_{Ci} < R_{Cj}$ then
7           Add Cj to the out-migration domain O
8           Add Ci to the in-migration domain I
9       Add $(C_i, C_j)$ to the migration queue Q
```

Now that we have our domains separated, we focus on optimally selecting switches to migrate between them. For this, we leverage switch migration efficiency and load rate reduction as two decisive factors for selecting switches. The ideal switch to migrate should reduce the maximum load ratio under the best migration efficiency. Also, each switch migration should reduce the discrete coefficient between the controllers. The constraints for selection of switches are as follows -

$$\begin{cases} R'_{C_x} = R_{C_x} \pm \sum_{k=1}^{l} L_{S_k} \times (C_i/C_j), \\ D_{(C_i,C_j)} = \left(\sqrt{\sum_{k=i,j} \left(R'_{C_x} - \overline{R_{C_i,C_j}} \right)^2 /2} \right) / \overline{R_{C_i,C_j}} \\ argmin\, D_{(C_iC_j)} \\ \forall E_{(C_i,S'_k)} \geq \overline{E_{(C_i,S)}} \\ \forall E_{(C_i,S'_k)} \geq \overline{E_{(C_i,S'_{k+1})}} \\ x = i \; or \; j \end{cases} \tag{6.1}$$

An implementation compiling the above discussion into code is presented below. The constraints listed above are followed to ensure optimal migrating switch selection.

Algorithm 2: Selection Algorithm of Migrating Switch

```
input  : Migration Queue $Q$
output: Switch collection S', migration tuples T
initialize: Set T to an empty set, n = 0
begin
1    foreach $q = \{C_i, C_j\} \in Q$ do
2        foreach $S_k \in C_i$ do
3            Calculate $E_{Sk}$
4            $E_{C_i,S} \longleftarrow E_{C_i,S} + E_{C_i,S_k}$
5            $L \longleftarrow E_{C_i,S_k}$
6        sort L
7        $\overline{E_{C_i,S}} \longleftarrow E_{C_i,S}/n$, $k = 1$
8        while $E_{C_i,S_k} \in L$ and $S_k \in S$ do
9            Calculate $R'_{Ci}$, $R'_{Cj}$ and $D'_{C_i,C_j}$
10           if $E_{C_i,S_k} \geq \overline{E_{C_i,S}}$ and $D'_{C_i,C_j} < temp$ then
11               k++, temp $\longleftarrow$ D'$_{C_i,C_j}$, add $S_k$ to S'
12           else
13               break
14       $T \longleftarrow S'$
```

6.3 Q-learning with ε-greedy

In this section, we are going to take a look at the proposed Q-learning algorithm and understand the theoretical aspects of the same before taking a look at the step by step implementation of the same.

6.3.1 Proposed Q-learning algorithm

As remarked before, our system moves through various states based on the arrival and departure of flows as well the actions we take at each decision timestamp. Let $Q_\pi(s,a)$ be the Q-value associated with a state s and an action $\pi(s)$ specified by the policy π. The aim is to determine an optimal policy π^* such that it maximised the Q-value associated with a state.

$$\pi^*(s) = \arg\max_{a \in A(s)} Q_\pi(s,a), \qquad \forall s, \pi \tag{6.2}$$

Let $a(n)$ be an update sequence which possesses the following properties,

$$\sum_{n=1}^{\infty} a(n) = \infty; \qquad \sum_{n=1}^{\infty} (a(n))^2 < \infty \tag{6.3}$$

Let b(n) be another update sequence which possesses the same properties as described in equation 6.3 along with the additional properties described below,

$$\sum_{n=1}^{\infty} (a(n) + b(n))^2 < \infty; \qquad \lim_{n \to \infty} \frac{b(n)}{a(n)} \to 0 \tag{6.4}$$

At every decision epoch, we update the Q-value associated with only one state action pair and keep the rest of the values unchanged. This scheme can be represented by the following equation. Note that here a(n) is the learning rate parameter typically represented by α and γ is the discount factor.

$$Q_{n+1}(s,a) = (1-a(n))Q_n(s,a) + a(n)\left[r(s,a) - l_n c(s,a) + \gamma \max a' Q_n(s',a')\right] \tag{6.5}$$

Here l_n is a Lagrange multiplier, which should be iterated along the timescale of b(n). Here S_n is given as a running sum of α raised to the power of iteration number, i.e., $S_n = 1.\alpha^0 + 1.\alpha + 0.\alpha^2 + ...$ The 1s and 0s are based on the number of switch exchanges on the iteration i.

$$l_n = A[l_n + b(n)(S_n - S_{max})] \tag{6.6}$$

Here A is a projection operator to keep the Lagrange multiplier between [0, L] for a large L.

6.3.2 Q-learning implementation

We use a ε-greedy strategy to train the Q-network by investigating a random action with probability ε and exploiting an action with probability 1 – ε that maximises the predicted long-term payoff. We acquire a new state and the resulting reward for each action we take. This is treated as a real-life experience that will be saved for later use. This procedure's algorithm is listed below.

Let's have a look at each of the steps we take in this algorithm.

- **Initialisation:** We first initialize the Q-matrix to all zeros. The hyper-parameters are set next. We set γ to 0.99, α to $\frac{1}{(n+1)^{0.6}}$, and $b(n)$ to $\frac{1}{n}$. We also initialise ε to 0.7. We then decrement this by 0.0025 per iteration

Algorithm 3: Load Balancing using Q-learning

```
input : $C$: set of all controllers
        $C_{cluster}$: initial arrangement of switches and controllers
        $load_{array}$: load on switches for any iteration i
        $n(s)$: number of switches in the SDN environment
        $n(C)$: number of controllers in SDN environment
output: cumulative discounted rewards
        discrete coefficients
        number of switch exchanges between controllers
begin
1   Initialize the evaluation matrix Q with all 0, $\gamma$, $\alpha$, $b(n)$, $\varepsilon$, $S_{max}$, $S_n$, and $l_0$.
2   foreach timesteps do
3       Determine the current system state (using last state and flows)
4       if exploration phase then
5           Choose one of the feasible actions at random
6       else
7           action = arg $\max_a$Q(s,a)
8       Observe reward r(s,a; $l_n) = r(s,a) - l_n c(s,a)$
9       Go to next state s'
10      Update Q(s,a) according to equation 6.5
11      Update the LM according to equation 6.6
12      Set current state to next state (s $\leftarrow s'$)
13      Record and update parameters for plotting graphs
14      if $\varepsilon > \varepsilon$-min then
15          $\varepsilon = \varepsilon * \varepsilon$-dec
16      else
17          $\varepsilon = \varepsilon$-min
```

till it reaches a minimum of 0.05. Finally, S_{max}, S_n, and l_0 are initialised.

- **State updation:** For each iteration we first update the state based on the incoming or outgoing flows. The state consists of number of flows on each controller as mentioned earlier.

- **Decide action:** Next we make a choice between exploration and exploitation based on ε value in that current iteration. Based on this, we decide upon one of the 3 actions. Unless the action is *do nothing*, the next state is different from the current state. We make necessary updations and store the next state.

- **Reward calculation:** Now that we have the next state, we make calculations for reward, cost and discounted reward. Update the Q-value for the current state-action pair. Finally we set the current state as next state and update ε value. Additionally in this step we make a note of the cumulative discount reward, current discrete coefficient and number of switch exchanges, which will be later necessarily for making plots.

Chapter 7

Results and Implementation

7.1 Experiment Setup

To evaluate the performance of the proposed schemes, we use python-based simulations. We use the Arnes network topology from the Internet Topology Zoo [1]. Along with the topology we make use of synthetic data generated from python code, which fits with our assumptions regarding the influx and servicing of flows in switches.

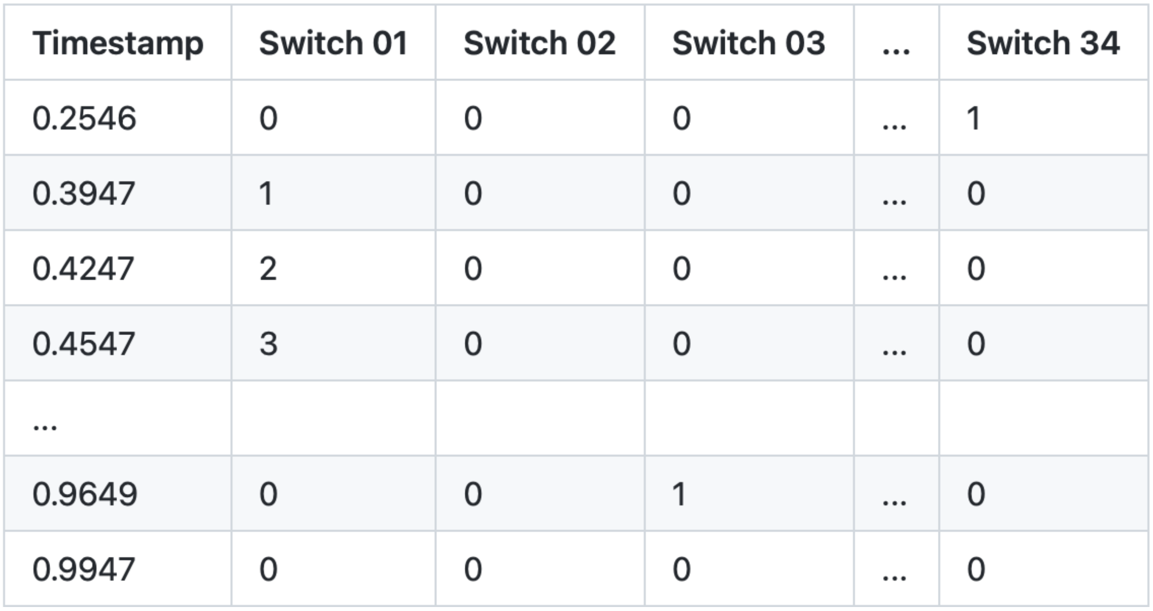
For data generation we come across the following 2 terms:

- **Arrival Rate (λ)** = We are assuming arrivals can be monitored using a Poisson process with parameter λ
- **Service Rate (μ)** = We are assuming that the packet service times cab be monitored using a Exponential process with parameter μ

For each switch we assume that the arrival is a Poisson process with parameter λ_i. However the service rate parameter μ is taken as for all switches. Basically since the arrivals occur as a Poi(λ) process, the inter-arrival times are exponentially distributed with mean λ. For each switch we compute the time of arrivals and add the service time to it to note the timestamp when the packets depart. It is interesting to observe that at each timestamp only one arrival or departure can occur. This means we can save tuples in the following format -

```
tuple(timestamp, switch, flow +/-)
```

Using these tuples we finally build our table of data. This is how our table will look roughly. For our experiment we have taken 34 switches (from the Arnes topology) and 5 controllers.



Timestamp	Switch 01	Switch 02	Switch 03	...	Switch 34
0.2546	0	0	0	...	1
0.3947	1	0	0	...	0
0.4247	2	0	0	...	0
0.4547	3	0	0	...	0
...					
0.9649	0	0	1	...	0
0.9947	0	0	0	...	0

Fig. 7.1 Example of a sample data table

7.2 Evaluation Metric:

- **Discrete Coefficient of Controllers:** The discrete coefficient between the controllers is defined as the degree of deviation of load between the controllers. A smaller discrete coefficient means better results.

$$D_{(C_i,C_j)} = \left(\sqrt{\sum_{k=i,j} \left(R_{C_k} - \overline{R_{C_i,C_j}}\right)^2 / 2} \right) / \overline{R_{C_i,C_j}}, \tag{7.1}$$

- **Cumulative discounted Reward:** The discounted reward is taken over all time points as a running sum. This reward should converge to a particular value in order for the Q-Learning to be considered valid.

$$r_{discounted}(n) = r(s,a) * \gamma^n \tag{7.2}$$

7.3 Experimental Results

We consider a system with $k = 5$ controllers and generate data for a variety of scenarios. To simulate real world environments we consider the following cases:

- **Heavy Load:** This is the scenario where the load on switches is more. This means that the arrival rate on the switches will be much higher than the service rate. In this case we consider the following rate of values to generate the data.
  - λ = random numbers in range (50, 100)
  - μ = a random number in range (1, 5)

- **Light Load:** This is the scenario where the load on switches is less. This means that the arrival rate on the switches will be much smaller than the service rate. In this case we consider the following rate of values to generate the data.
  - λ = random numbers in range (1, 5)
  - μ = a random number in range (50, 100)
- **Skewed Load:** This is the scenario where the load on switches is skewed. This means that the arrival rate on the switches will be much higher than the arrival rate on other switches. For our purpose let's put a very high arrival rate on randomly chosen 2 switches. In this case we consider the following rate of values to generate the data.
  - λ = high: random numbers in range (50, 100); low: random numbers in range (5, 10)
  - μ = a random number in range (25, 50)

The figures for all 3 cases are shown below. We make the following **observations**:

- The cumulative discounted reward converges to a specific value in all cases. This happens after roughly 400 iterations and is a good sign because this validates that the Q-Learning is functioning effectively.
- The discrete coefficient is much more random and higher in case of light load than in the case of heavy and skewed load as shown in Fig 7.5. In the latter cases, we notice that the discrete coefficient gradually decreases till it stays roughly in a small range, which means that the system load has stabilized and is equally distributed.

- The number of switches exchanged is plotted cumulatively, and we notice that it is smaller in case of light load. For heavy and skewed systems this number reaches around 4000 which is roughly close to the chosen S_{max}.
- For heavy load scenario, we in Fig 7.2 see that the discrete coefficient is in the range of 0.25-0.5 roughly. This is a good value, since the system is constantly being disturbed by new arrivals and departures or flows. We also see that the cumulative discounted reward converges to -55 roughly around 400 iterations. After that the change is very small to be visible on the curve.
- For light load scenario, we see that the discrete coefficient is random. This is justified by the fact that the discrete coefficient here is a measure of the deviation of the number of flows in each switch. Since flows are serviced very quickly, the system usually stays in a state where load is on a few controllers only and the others have 0 load. This gives a large D value.
- For skewed load scenario in Fig 7.8, we observe that the discrete coefficient values are more random that the heavy load scenario. This is also backed by the fact that in case of skewed load, inflow is constantly more on a few switches and very less on others.

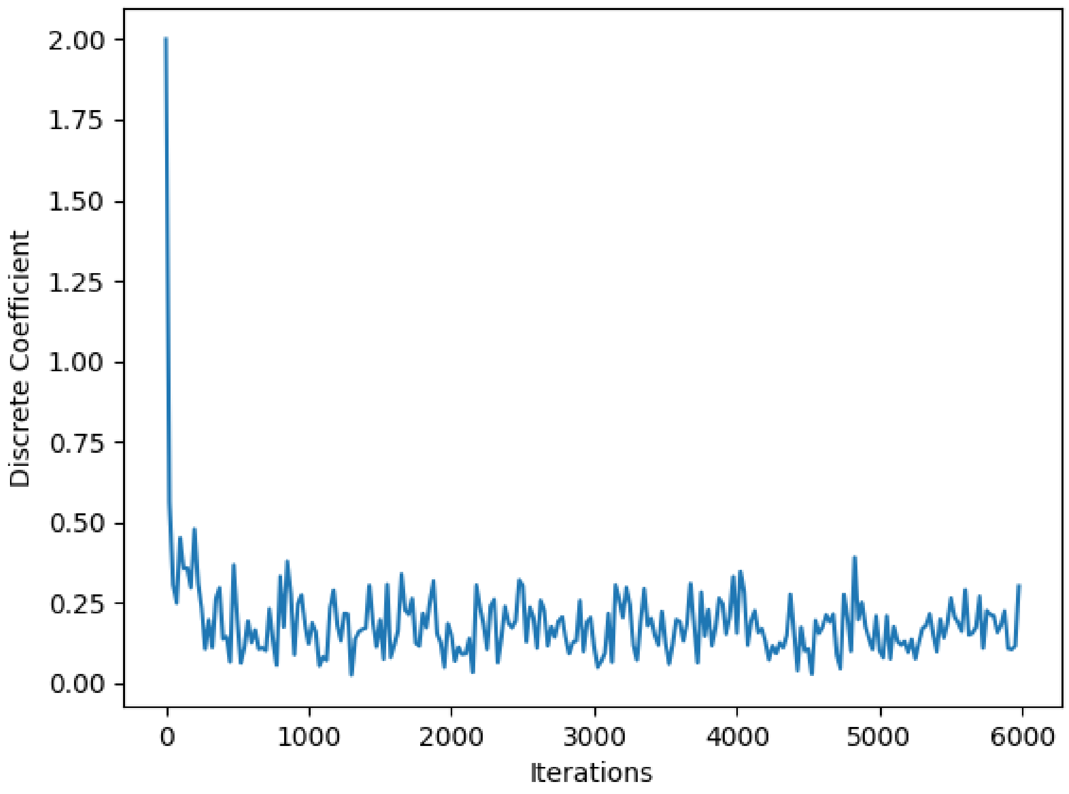


Fig. 7.2 Discrete coefficient of the system with time - Load Heavy

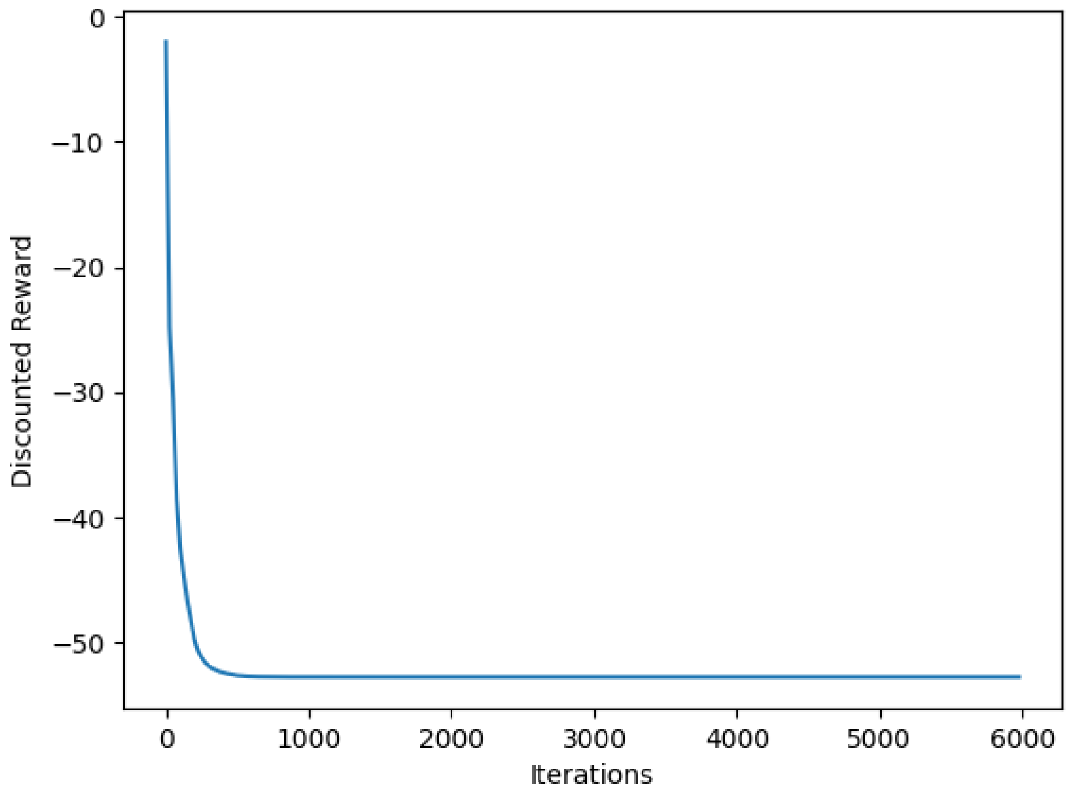


Fig. 7.3 Cumulative discounted reward of the system with time - Load Heavy

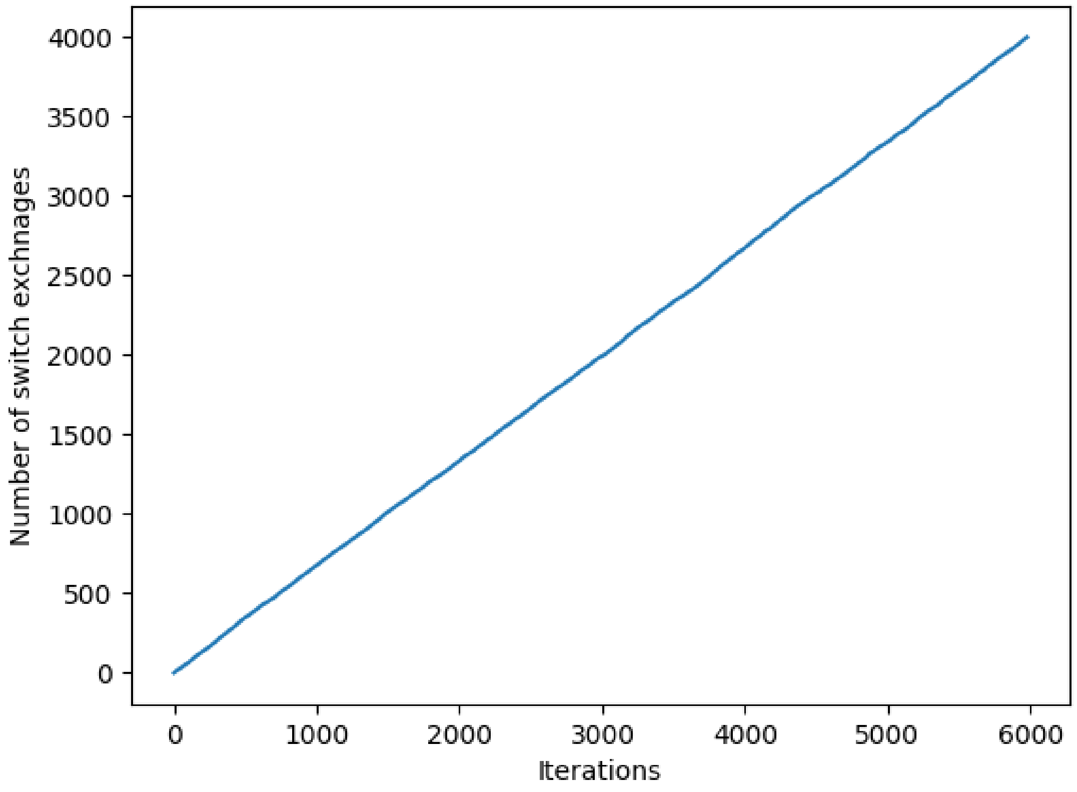


Fig. 7.4 Number of switches exchanged with time - Load Heavy

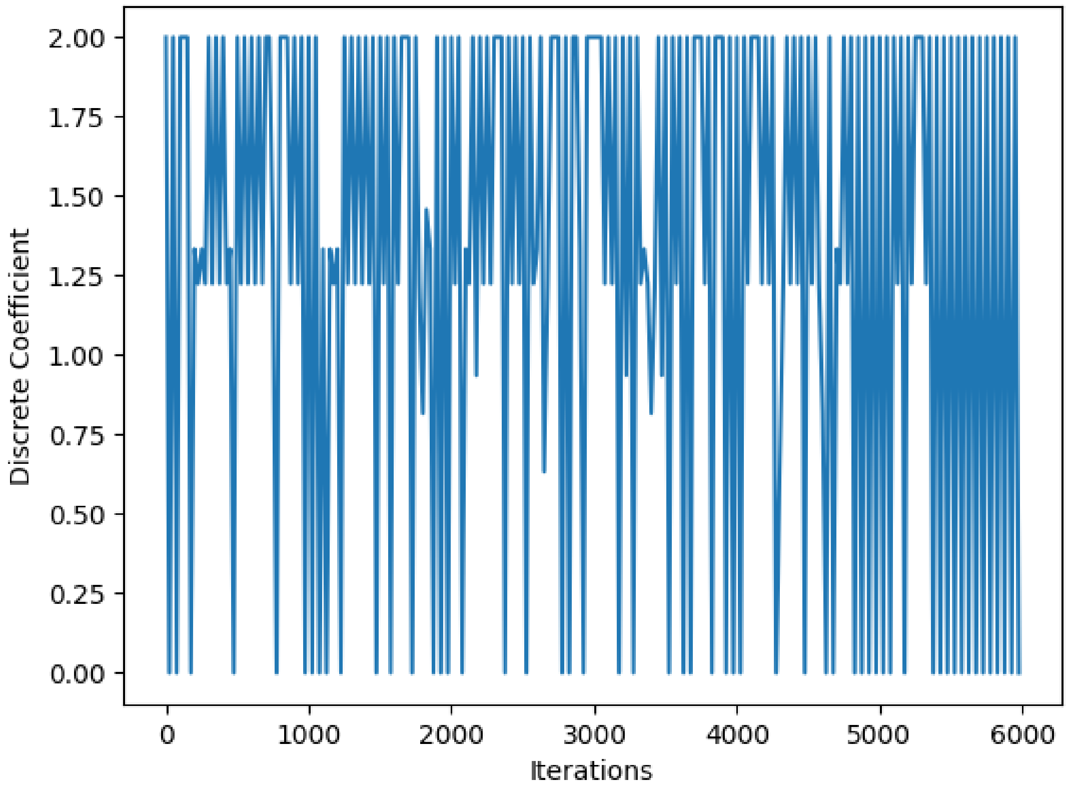


Fig. 7.5 Discrete coefficient of the system with time - Load Light

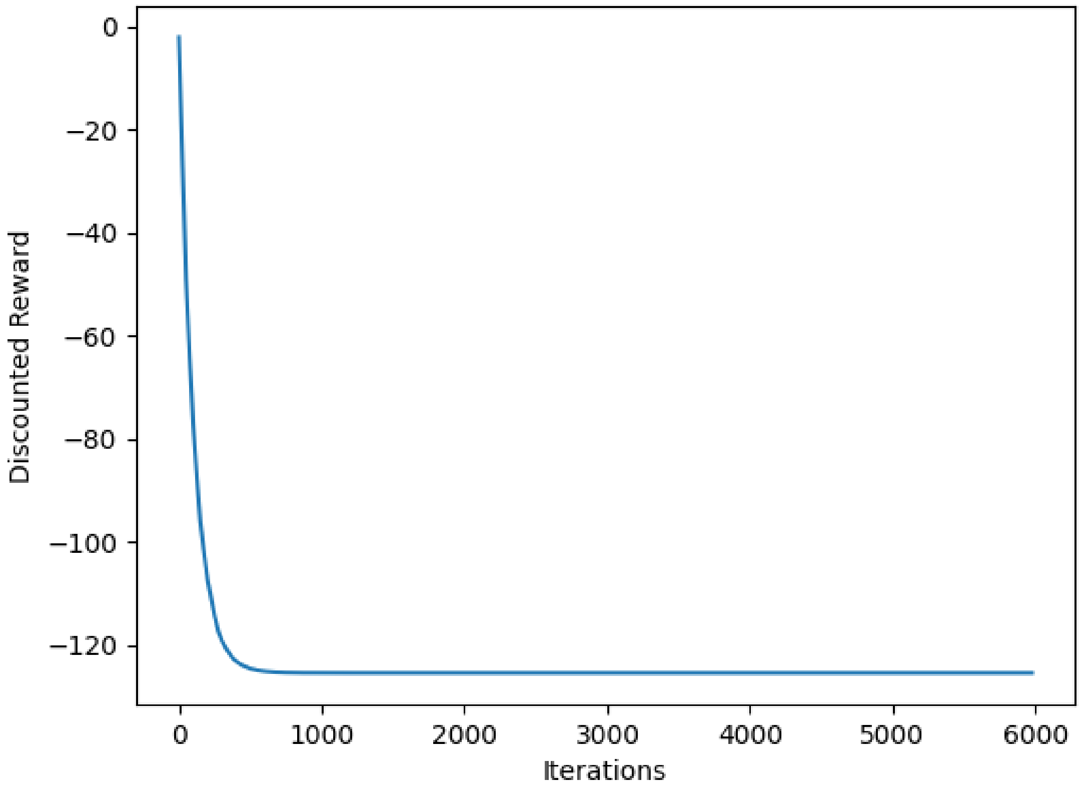


Fig. 7.6 Cumulative discounted reward of the system with time - Load Light

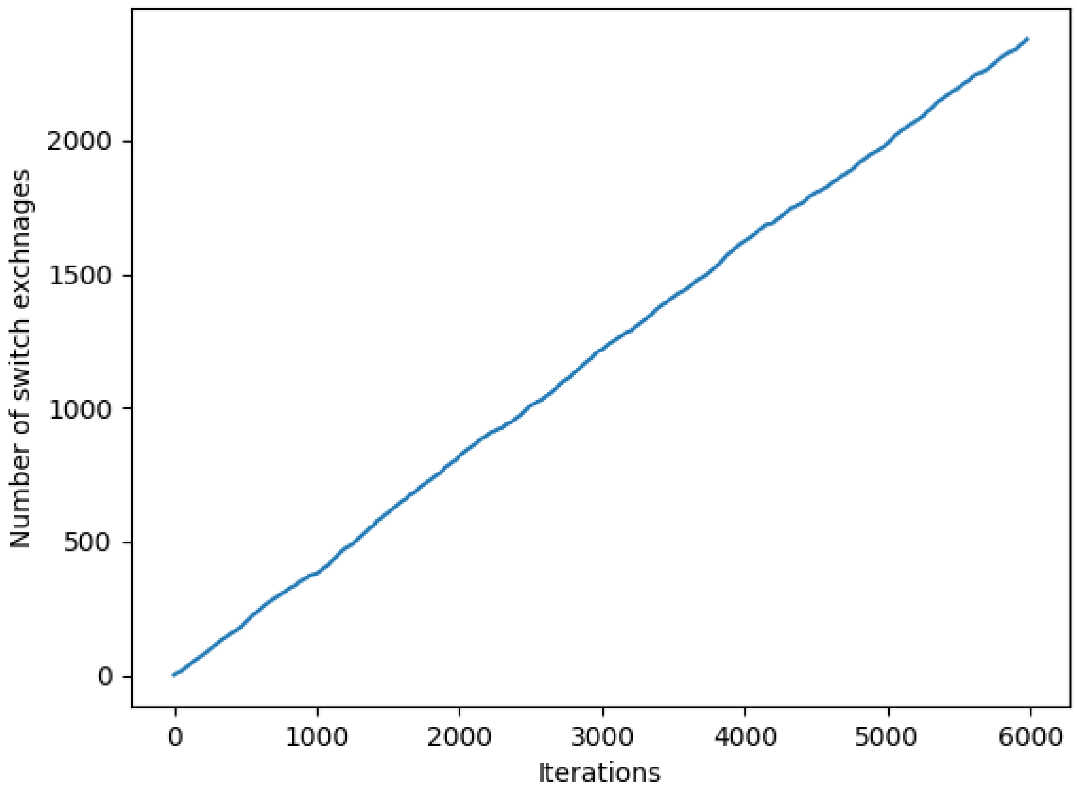


Fig. 7.7 Number of switches exchanged with time - Load Light

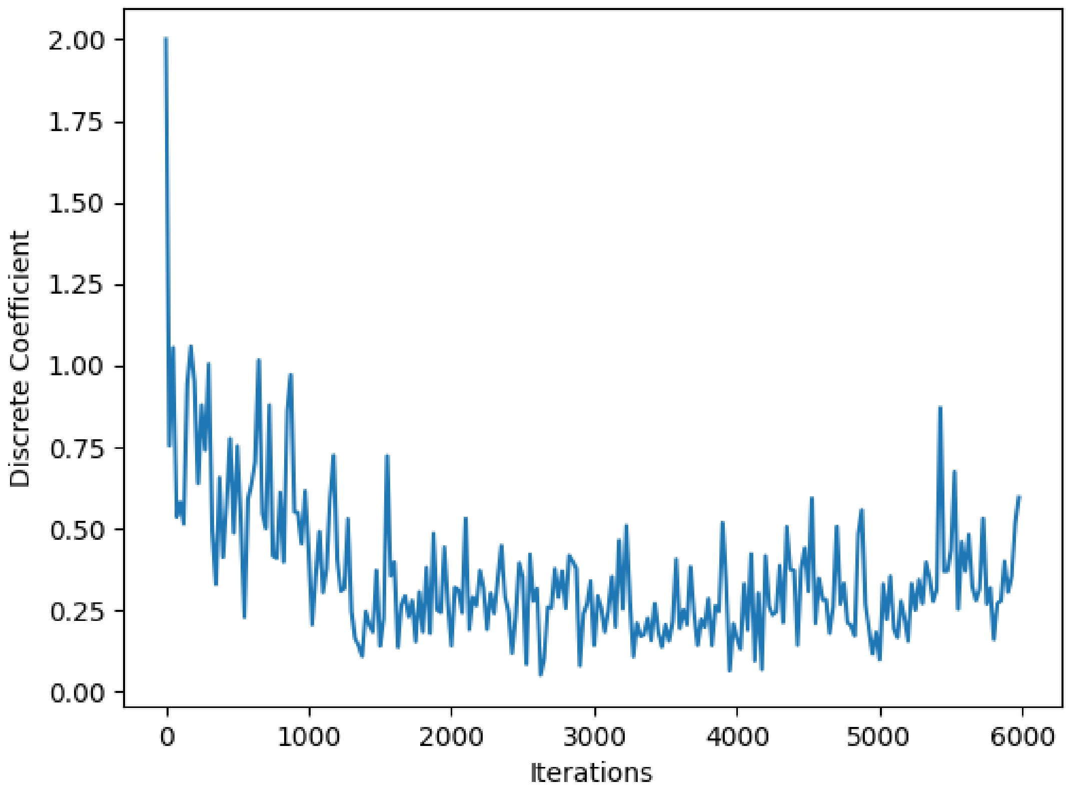


Fig. 7.8 Discrete coefficient of the system with time - Load Skewed

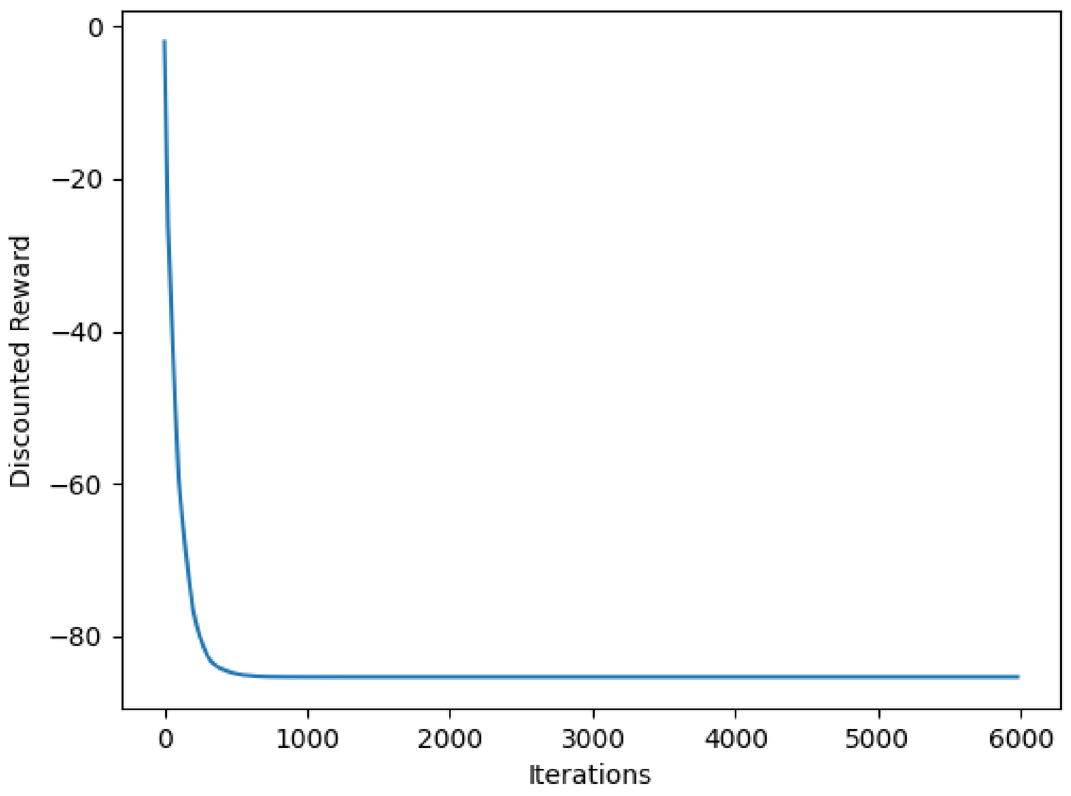


Fig. 7.9 Cumulative discounted reward of the system with time - Load Skewed

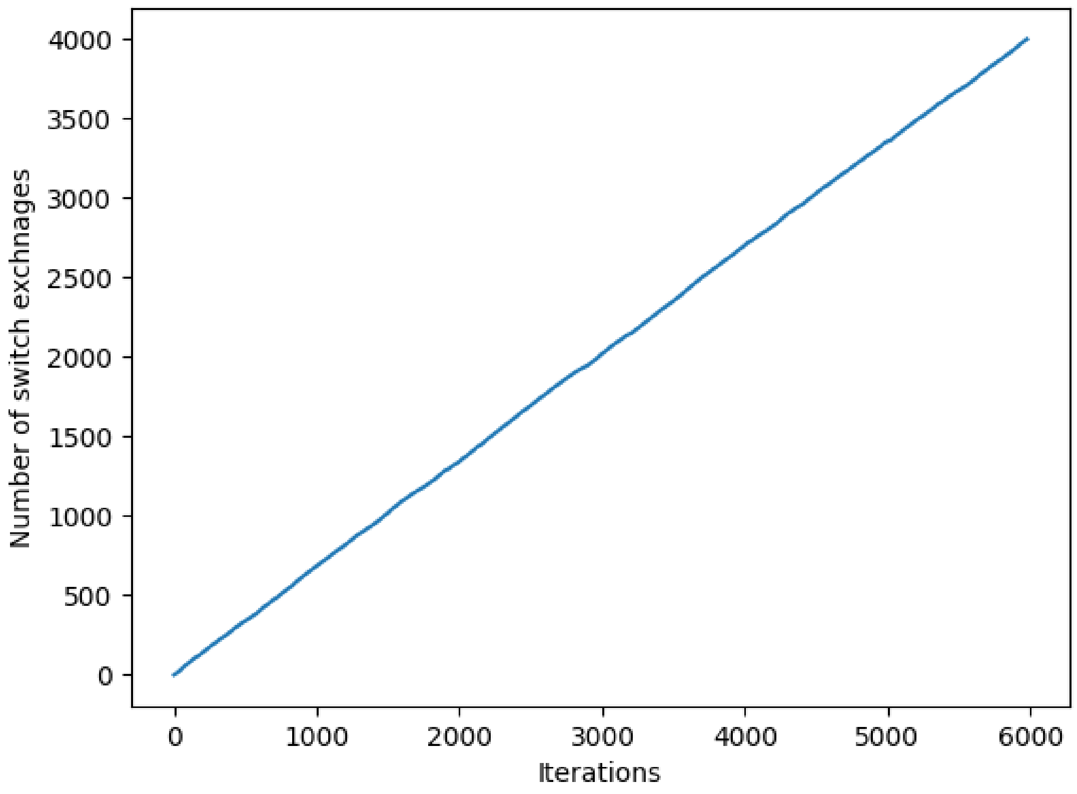


Fig. 7.10 Number of switches exchanged with time - Load Skewed

Chapter 8

Conclusion & Future Work

This report focused on SDN Controller load balancing with real traffic networks using reinforcement learning. The results obtained have motivated us to extend this work. Our next goal would be work on real data obtained from different sources and also explore other alternatives such as deep Q-learning to capture the network traffic behavior to balance the controller and efficient switch migration considering dynamic networks.

Bibliography

[1] The internet topology zoo. http://http://www.topology-zoo.org/index.html, 2019 (accessed January, 2020).

[2] H. Chen, G. Cheng, and Z. Wang. A game-theoretic approach to elastic control in software-defined networking. *China Communications*, 13(5):103–109, 2016.

[3] Advait Dixit, Fang Hao, Sarit Mukherjee, TV Lakshman, and Ramana Kompella. Towards an elastic distributed sdn controller. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 7–12, 2013.

[4] Advait Dixit, Fang Hao, Sarit Mukherjee, TV Lakshman, and Ramana Rao Kompella. ElastiCon; an Elastic Distributed SDN Controller. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 17–27, 2014.

[5] A. Filali, S. Cherkaoui, and A. Kobbane. Prediction-based switch migration scheduling for sdn load balancing. In *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, pages 1–6, 2019.

[6] Tao Hu, Julong Lan, Jianhui Zhang, and Wei Zhao. EASM: Efficiency-Aware Switch Migration for Balancing Controller Loads in Software-Defined Networking. *Peer-to-Peer Networking and Applications*, 12(2):452–464, 2019.

[7] Diego Kreutz, Fernando M. V. Ramos, Paulo Esteves Veríssimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2015.

[8] Abha Kumari and Ashok Singh Sairam. Controller placement problem in software-defined networking: A survey. *Networks*, 2021.

[9] Taeha Kim Sangho Yeo, Ye Naing and Sangyoon Oh. Achieving balanced load distribution with reinforcement learning-based switch migration in distributed sdn controllers. *Electronics*, 10(162):1–16, 2021.

[10] Ashutosh Kumar Singh and Shashank Srivastava. A survey and classification of controller placement problem in sdn. *International Journal of Network Management*, 28(3):e2018, 2018.

[11] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[12] Md Tanvir Ishtaique ul Huque, Weisheng Si, Guillaume Jourjon, and Vincent Gramoli. Large-Scale Dynamic Controller Placement. *IEEE Transactions on Network and Service Management*, 14(1):63–76, 2017.

[13] Chuan'an Wang, Bo Hu, Shanzhi Chen, Desheng Li, and Bin Liu. A Switch Migration-Based Decision-Making Scheme for Balancing Load in SDN. *IEEE Access*, 5:4537–4544, 2017.

[14] Yaning Zhou, Ying Wang, Jinke Yu, Junhua Ba, and Shilei Zhang. Load Balancing for Multiple Controllers in SDN based on Switches Group. In *19th Asia-Pacific on Network Operations and Management Symposium (AP-NOMS)*, pages 227–230. IEEE, 2017.